The Derivation Pipeline

Introduction

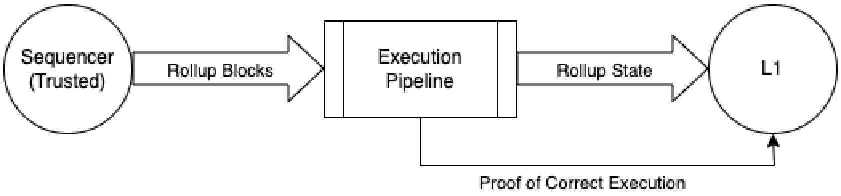
In previous posts, we described how the Espresso Sequencer can be used both to decentralize rollups while retaining many of the UX benefits of centralized sequencers, and to improve interoperability between rollups, helping to defragment liquidity in a world with many different L2s. With the recent public launch of our Cortado testnet, we have taken another step towards realizing this vision, by running two different rollups from two different stacks (OP Stack and Polygon zkEVM) on the same decentralized sequencer.

With these two different stack integrations under our belt, we will begin sharing more details about the architecture of a typical integration between a rollup and the Espresso Sequencer. This post will focus on a crucial and under-discussed part of the architecture: the derivation pipeline, which connects the rollup execution layer to the sequencer.

We will define the derivation pipeline and share how rollups can easily deploy on the Espresso Sequencer and customize their own derivation pipeline. We will then describe how the derivation pipeline can be used to add rollup features on top of the features provided natively by the Espresso Sequencer, more easily than by modifying the rollup's execution layer. As an example, we describe how rollups can support powerful cross-chain atomic transaction bundles by modifying their derivation pipelines and leveraging the Espresso Sequencer's atomic inclusion features.

The Derivation Pipeline

Let's compare the architecture of a rollup running on the Espresso Sequencer with the typical centralized-sequencer architecture of today's rollups, which looks like this:

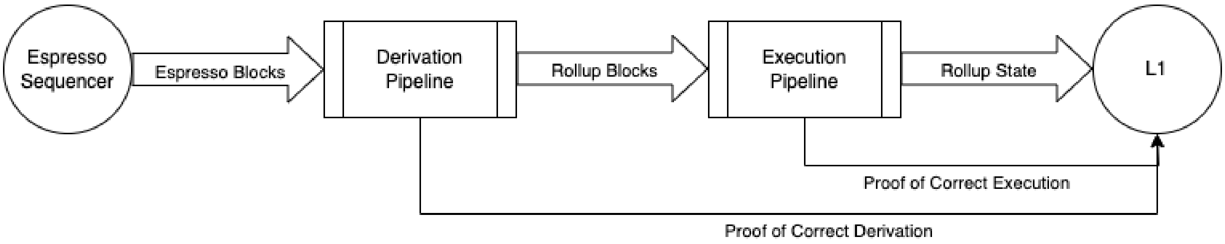


A sequencer, usually a centralized server operated by a foundation or the company that built the rollup, produces a stream of rollup blocks: lists of transactions and other metadata. The execution pipeline executes these blocks by applying the rollup's state transition function to the block stream in order to compute and update some state. This state is then sent to another blockchain, the layer 1 or L1. Since the execution pipeline runs offchain, the L1 needs some sort of proof that the rollup state was computed correctly. This is typically done either by requiring the node that ran the execution pipeline to send a succinct proof to the L1 showing that it ran the pipeline correctly, or by optimistically treating the state as correct until some other node sends a proof that the execution wasn't done correctly.

In this architecture, the sequencer has a lot of power. Depending on the specific rollup design, it may be able to adversarially impact the transaction order (e.g., sandwich attacks), censor certain transactions, and stall liveness. The sequencer generally cannot affect the correctness of the rollup state, since the computation of the state is verified by the L1. However, in practice, many users trust the sequencer with an additional responsibility: finality. Users often consider their transaction finalized as soon as it has been emitted by the sequencer. However, the transaction is not really final until it has been received by the L1. This means the sequencer can lie about a transaction being finalized, then send a different ordering which excludes that transaction to the L1. Akin to a fork in a traditional blockchain, this can lead to double spends and loss of funds.

To address this security risk, we replace the centralized sequencer with a consensus protocol running on thousands of nodes. Anyone can submit a state update to the L1 contract as long as it is consistent with the output of this consensus protocol, which in the case of the Espresso Sequencer is called Hotshot. This both decentralizes the rollup and removes the ability of any coalition of nodes controlling less than ⅓ of the Espresso Sequencer staking power to equivocate on the rollup's transaction ordering. Hotshot is highly available and Byzantine-fault tolerant, which essentially means that it will remain live and will not equivocate on finality unless a large amount of economic collateral is controlled by a malicious party. It is also possible for rollups to impose additional constraints on transaction ordering. For example, this allows rollups to optionally enforce first-come-first-served sequencing or choose from a variety of approaches to preventing sandwich attacks and the like.

We need the rollup itself to enforce the constraint that the sequence of blocks it is executing is determined by the output of Hotshot which ultimately means that the L1 will need to check this constraint. Thus, we add a second pipeline, which feeds into the execution pipeline, whose job is to derive the rollup block stream from the output of the Espresso Sequencer, and to produce some kind of proof that this derivation was done correctly. This is the derivation pipeline.



Note: the derivation pipeline is already a feature of rollups today and a term that is used in the OP stack. Today it is used to "permission" the sequencer, i.e. check that the state update posted to the L1 was signed by the designated sequencer node.

Let's look at what happens to the Espresso block stream when it passes through the derivation pipeline for a particular rollup.

Consensus Verification

The first thing the derivation pipeline must do is prove that the blocks in its input are the sequential outputs of the Espresso Sequencer. Luckily, the HotShot consensus protocol underlying the Espresso Sequencer produces verifiable artifacts proving the finality of each block it outputs. These artifacts are called quorum certificates, or QCs.

A QC is essentially a threshold signature by the participants of the consensus protocol attesting to the finality of a block. Therefore, the most direct way of proving that a block is the next output of the sequencer is to incorporate a proof of correct signature for the appropriate QC into the proof of correct derivation that is sent to the L1. There is, however, an easier way.

Since the Espresso block stream is not specific to any one rollup, the work required to verify it is shared across all rollups. Therefore, the Espresso Sequencer will include, as a public good, an L1 smart contract whose sole job is to verify QCs and provide a certified commitment to the Espresso block stream in L1 storage. Since the proof of correct derivation is verified on L1 anyways, the rollup's L1 contract can read directly from this certified commitment. The derivation pipeline then only needs to prove that its input block stream is consistent with this commitment, which it can do via Merkle proof.

Namespace Filtering

The next job of the derivation pipeline is to convert the Espresso block stream, which includes transactions from all the rollups using the Espresso Sequencer, into a rollup-specific block stream, which includes only transactions for this particular rollup. This is done via *namespace filtering*: every rollup has a namespace identifier, which is like a chain ID, that identifies its transactions. The transactions in an Espresso block are grouped by namespace, so the rollup can easily download just those transactions that belong to it by requesting the transactions from the appropriate namespace from an Espresso Sequencer query service.

Of course, the rollup must be able to verify that the query service has given it the right transactions, and in turn the derivation pipeline must prove this same fact to the L1 contract as part of the proof of correct derivation. To facilitate this, the query service will provide, along with the requested transactions, a namespace proof. This proves, with respect to an Espresso block commitment, that the provided list of transactions includes all the transactions in the block from the requested namespace, and only those transactions, in the order determined by the Espresso sequencer. This proof is efficient to verify in hardware, and verification is also arithmetic circuit-friendly for use in ZK proofs.

The derivation pipeline and ultimately the rollup contract will verify this proof relative to an Espresso block commitment, which was in turn verified to be a valid output of the Espresso sequencer in the previous step. The exact design of the namespace proof is complex and is closely related to the erasure coding scheme used in Espresso's Savoiardi DA protocol. We will share more details on the namespace proof protocol in a future blog post.

Rollup-Specific Transformations

As explained, the derivation pipeline thus allows every individual rollup to identify and verify its unique blockstream among the entirety of data being processed by the Espresso Sequencer. And because the derivation pipeline is logically separated from the execution pipeline and other aspects of the rollups internal functioning, they can choose to interpret the Espresso blockstream however they want.

For example, in theOP stack integration, the OP stack rollup conforms to a fixed blocktime, as this is a constraint built into the OP stack itself. In the derivation pipeline, the OP stack can easily derive a fixed blocktime chain from Espresso's variable block time chain by grouping together Espresso blocks within each fixed-size window of time. This requires no changes to the OP Stack execution layer; only a modification to how the OP block stream is derived from the Espresso block stream.

Meanwhile, the Polygon zkEVM stack integration does no transformations and interprets the blockstream as is. So rollups can also just take the blockstream and feed it into their execution pipeline at 1:1 value.

The derivation pipeline also enables encrypted mempools, fair ordering services such asTimeboost, designated builders, and other design constraints that may be specific to individual rollups, while still being easily supported by the Espresso Sequencer.

Atomicity Through the Pipeline

While the derivation pipeline is used to split up the Espresso Sequencer's blockstream into smaller blockstreams unique to each individual rollup, the sequencer is still including transactions from multiple rollups in the same block.

We can leverage the pipeline to provide atomicity for transactions originating from multiple rollups. An atomic cross-chain bundle is a set of transactions that is only valid when included on all rollups in the same block but not if it is split up. For example, this can be two transfers on two separate rollups. Such bundles enable atomic swaps between tokens on different rollups. The true power comes when atomic bundles are combined with a shared builder or some bridge mechanism. These can enable entire new classes of applications such as atomic bridges or cross-chain flash loans (see this talk for more detail).

Signing atomic bundle transactions

Atomic bundles have the key feature that they are inseparable, i.e. no transaction within the bundle is valid on its own. A simple way to achieve this is if the issuer of the transaction bundle signs it as a whole and the sequencer pipeline ensures validity of the transaction signature. The signature verification is part of the transaction execution, i.e. is verified within the rollups themselves. This presents a challenge as each rollup only sees the filtered transactions relevant for that rollup. A naive implementation of atomic x-rollup bundles would break backwards compatibility with individual rollups. Consider the following strawman proposed by James Prestwich:

There is a new transaction format for bundle transactions and the user signs the entire rollup bundle. Now rollup nodes can verify whether a bundled transaction is valid by computing the hash of the bundle and checking the signature on this bundle hash. Additionally, the user needs to strip signatures from the transactions within the bundle. This is important as otherwise a malicious builder could break up the bundle and submit the transactions individually, which would break the atomicity guarantee that we are trying to achieve.

This breaks backwards compatibility with the way that rollups are currently implemented, as rollups would need to modify their execution pipelines to process bundle transactions stripped of their individual signatures.

We recently described a similar scheme in thispost, that achieves the same atomic bundling but is backwards compatible, i.e. does not require opt in from the rollups. The idea is that bundled transactions add an ephemeral bundle public key to the individual transactions themselves. This bundle public key is signed as part of the normal transaction signing. We do this by appending the public key to the call data of the transaction such that it doesn't change the meaning of the transaction. The public key can then be used to authenticate the bundle.

Concretely the protocol works as follows:

- Each bundle transaction adds the following string to the call data of the transaction
  - MAGIC_STRING BUNDLE_PK
- A sequencer block includes the bundle of transactions, the list of rollups for this bundle, and a signature under BUNDLE_PK on the hash of those fields
- The block header contains a namespace table, which points to this bundle for each namespace involved in the bundle

The derivation pipeline is changed as follows:

- For each transaction (or bundle) within the rollups namespace the rollup's pipeline checks that
  - For a normal transaction there is no magic string in the calldata
  - There exists a signature under BUNDLE_PK on the hash of the transaction bundle, including all the namespaces
  - The pipeline checks that all the bundle transactions are included in all rollups.

The wrapper format of the bundle transaction would have this form:

```

namespace A

namespace B

signature(namespace A, namespace B, txA', txB') under pk

txA' (with pk appended)
```

txB' (with pk appended)

```

The workflow for generating a transaction can be best visualized using a central coordinator (e.g. a builder). However this coordinator can be replaced with a simple peer to peer protocol:

- Each user randomly generates an ephemeral key pair and sends pk to a coordinator
- Coordinator responds with an aggregate pk after all parties have generated key pairs
- Each user builds their transaction and appends the aggregate pk to the calldata, and signs it with the rollup transaction key
- Each user sends the transaction to the coordinator
- Coordinator responds with an unsigned bundle template once all parties have sent their signed transactions
- User signs the bundle with the ephemeral key and sends this signature to a builder
- Builder aggregates all signatures and publishes the signed bundle

#### Properties

A key property of the scheme is that transaction bundles are atomic and cannot be split apart and individually submitted. This is ensured through the bundle public key and the signature which signs both the transactions as well as the rollups in which the bundle is supposed to be included. The other important property that the scheme achieves is backwards compatibility. This is achieved through appending the public key to the calldata. Normal smart contracts will simply ignore this additional calldata, as the decoding of method arguments is invariant under length extension using the standard ABI encoding.

#### Applications of cross-rollup atomic transactions

We laid out the applications of cross-chain atomicity previously inthis blog. Cross-chain atomic transactions have a few direct applications such as atomic swaps, i.e. swapping an asset on one rollup with another asset on a different rollup. The most powerful use case of cross-chain atomicity is that it enables builders to construct blocks with atomic bundles. These bundles cannot be torn apart even by a malicious proposer.

Builders can use atomic bundles to provide a guarantee that a user’s transaction will only be included in the Espresso Sequencer if it meets some predetermined execution criteria. The shared sequencer is what allows builders to give such a guarantee, because the builder can be in full control of the state of all rollups they are including transactions from. This means that an honest builder will never deviate from their promises. We can even build out new classes of applications, such as cross-chain flashloans, that rely on the atomicity guarantees provided by the builder.

This is opposed to a world with multiple sequencers, where a builder could provide a guarantee to users, but fail to get their blocks included by the individual sequencers. Additionally, in this case there is no way of knowing whether the builder was malicious, or simply failed to convince each sequencer to include their block. This means that atomicity guarantees leveraging an economic penalty are incompatible with siloed sequencers, because an honest builder could still arbitrarily fail to stick to their promises. Furthermore, the success rate of cross-chain atomicity will be unpredictable, leading to worse economic outcomes.
```